

Overview

The purpose of this document is to provide a brief introduction to the template language implemented in UMA. It assumes you have a working knowledge of the object/property metamodel of UMA.

It is not intended to provide an exhaustive explanation of the architecture. UMA team members can provide you with such an explanation and they can direct you to many code samples that use this facility.

The purpose of the UMA Template Language (UTL) is to provide scripting capabilities to any UMA-based application. These capabilities can be used for various features, including (but not limited to):

- **Macro expansion:** Runtime expansion of code templates against the model. Examples include trigger code and stored procedures in OR•Compass, and Visual Basic and Java components in SQL•Compass.
- **Scripting:** Execution of a series of commands against the model. An example is the controller code for Forward Engineering in OR•Compass.

The architecture of UTL is intended to promote its use in as many aspects of an UMA-based product as possible. This is accomplished by allowing the user¹ maximum flexibility in tailoring the language to the specific needs of his product, while providing as much implementation support as possible and exposing mechanisms that allow reuse of new implementation code.

To accomplish these goals, UTL:

1. defines a minimal syntax for the language;
2. allows the user to write procedures and iterators;
3. provides a mechanism for registering the procedures and iterators in registries.

Syntax Summary

The following is a brief summary of the syntax of UTL intended to provide the bare minimum necessary to understand the implementation. More information can be found in the documentation shipped with OR•Compass 1.0.

Template code consists of literals, macros, comments and a few operator/scoping characters. Currently, UTL views all code as strings: numeric values are described by their string representation; most macros evaluate to strings. etc.²

¹ In this document, "user" refers to the UMA application developer, not the end-user of the application.

² UTL 1.5, expected this fall, will also support numeric modes of operation that will provide better performance when dealing with numbers. This will be useful for features such as volumetric calculations.

Literals

In template source code, all text outside of curly braces is treated as a literal, and is emitted exactly as typed. Within curly braces, text inside double quotes is treated as a literal. The former allows large blocks of boilerplate to be entered, e.g. the body of a stored procedure. The latter allows you to embed literals within macro calls.

Example:

```
This text would emit just like this.
{"So would this text."}
```

Macros

Macros are special instructions to the macro processor...basically procedures. All macros are contained within curly braces. Macros basically fall into two categories: procedure macros and iterator macros.

Procedure macros are designed to perform some work. They may expand to a value; they may declare a variable; they may invoke a process. The action(s) performed are entirely specified by the designer of the macro. The only constraint upon them is that they return True upon successful completion of their task and False upon failure.

Example—the following shows a string literal, followed by a macro call for getting the page number when printing:

```
My Model Report - Page {HeaderPage}    ⇨ Input
My Model Report - Page 1                ⇨ Output
```

Iterator macros allow the user to traverse across data structures. Iterators are distinguished by the keywords begin and end that delimit a block of code following the iterator declaration. All of the code within this begin/end block will be executed once for each iteration. When the iterator has moved across all objects in its pool, control will break out of the iteration block and execute the first statement after the block.

Example—the following executes the procedure macro DoSomething once for each element returned by the MyIterator macro. Note the curly braces surrounding the entire code fragment...indicating that all code within is to be treated as macro code.

```
{
  MyIterator
  begin
    DoSomething
  end
}
```

Macros, both procedure and iterator, can take parameters. Parameters are strings, or anything that evaluates to a string, such as another macro. If a macro takes parameters, they are enclosed in parentheses after the macro. Since forward declaration of macros is not required, macros may accept variable-length parameter lists, if desired.

Example:

```
{ MacroWithParameters("foo") }
```

Control blocks

Occasionally, it is desirable to have an entire block of template code fail if any piece of it fails. An example would be where there was some boilerplate text followed by an optional value; if the value was not present, the boilerplate should not be emitted.

Any code enclosed in square brackets is *conditional*: it will all fail if any piece of it fails. "Failure" means that a macro will not execute and a literal will not be emitted.

Example—if there is no middle name, the period will not emit:

```
{ FirstName [MiddleInitial "."] LastName }
```

Conditional blocks hide their internal scope. This means that the failure of a conditional block has no effect on the surrounding code. Any code enclosed in angle brackets is *propagating conditional*: it will fail if any piece of it fails and it will propagate that failure out to the next block.

The best way to distinguish these two types of blocks is with examples. In both examples, the Print macro emits the word "foo" and the Fail macro fails no matter what.

Example of conditional block:

```
{ Print " " [ Print [ Fail ] ] }    ⇐ Input
foo foo                             ⇐ Output
```

Example of propagating condition block:

```
{ Print " " [ Print < Fail > ] }    ⇐ Input
foo                                 ⇐ Output
```

Architecture

LWMDDataButler

The desired capabilities of UTL are achieved using an object called a data butler. Data butlers are responsible for providing implementations for the macros encountered by the macro processor when executing template code. When the macro processor is invoked, a data butler interface³ is supplied to it. Essentially, a data butler is a use of the Strategy pattern.⁴

³ Interface defined in LWMDDataButler.h

⁴ See Gamma, Helm, Johnson Vlissides, Design Patterns

This accomplishes the first goal of the language design. By providing different data butlers to the macro processor, the grammar of UTL can be configured for the application and task at hand. Each data butler implementation can accept a set of keywords (macros) appropriate for its task and reject keywords that are not acceptable. Further, a given keyword can be implemented in different ways by different data butlers, allowing template code to be ignorant of the underlying implementation. This is useful in allowing multiple products to work from the same templates when their underlying data models differ.

The LWMDDataButler interface has five entry points:

```
// Handle procedure macro with no parameters
virtual LWTBoolean SubstituteValue(
    const LWTString & MacroName,
    LWTString & ExpansionValue) = 0 ;

// Handle procedure macro with parameters
virtual LWTBoolean SubstituteValue(
    const LWTString & MacroName,
    LWMStringList & ParameterList,
    LWTString & ExpansionValue ) = 0 ;

// Handle iterator macro with no parameters
virtual LWTBoolean StartIteration(
    const LWTString & IterName ) = 0 ;

// Handle iterator macro with parameters
virtual LWTBoolean StartIteration(
    const LWTString & IterName,
    LWMStringList & ParameterList ) = 0 ;

// Handle iterator macro iteration
virtual LWTBoolean NextIteration(
    const LWTString & IterName ) = 0 ;
```

MCDDataButlerI

The second design goal of UTL was to provide as much functionality as possible to the user and to allow them to publish their own code for reuse.

Since the metaphor of an UMA model deals with all data at a very abstract level—objects and properties—many grammatical elements of a template language can be written abstractly. These elements can be aggregated together to form the major portion of the language needed by any UMA-based product. The user could extend this base in the few areas that are product-specific. Ideally, the language should be extendable not only by the user, but by the end-user of the user's product. This is accomplished in our solution.

There are two basic problems that are addressed by the solution:

1. A mechanism is needed to share implementations. As new keywords are added, they must be published.

- ⁶ *Op cit.*, Design Patterns

MyTable1 MyColumn1 MyColumn2 MyTable2 MyColumn3 ⇐ Output